

# Ordenação Baseada em Árvores de Fusão

Rogério H. B. de Lima

Luis A. A. Meira

Instituto de Ciência e Tecnologia

Faculdade de Tecnologia

Unifesp

Unicamp

7 de outubro de 2014

## Resumo

O problema da ordenação é sem dúvida um dos mais estudados na Ciência da Computação. No escopo da computação moderna, depois de mais de 60 anos de estudos, ainda existem muitas pesquisas que objetivam o desenvolvimento de algoritmos que solucionem uma ordenação mais rápida ou com menos recursos comparados a outros algoritmos já conhecidos. Há vários tipos de algoritmos de ordenação, alguns mais rápidos, outros mais econômicos em relação ao espaço e outros com algumas restrições com relação à entrada de dados. O objetivo deste trabalho é explicar a estrutura de dados Árvore de Fusão, responsável pelo primeiro algoritmo de ordenação com tempo inferior a  $n \lg n$ , tempo esse que criou certa confusão, gerando uma errada crença de ser o menor possível para esse tipo de problema.

## 1 Introdução

O problema da ordenação é talvez o mais estudado da Ciência da Computação. A sua utilização está implícita em etapas intermediárias de quase todos os programas existentes, como banco de dados, planilhas, multimídia etc. Além disso, a ordenação é estudada pela computação há mais de setenta anos. O algoritmo *Merge Sort*, largamente utilizado nos dias de hoje, foi proposto por Von Neumann em 1945 [5].

O problema da ordenação consiste em receber uma sequência de  $n$  números como entrada  $A = (a_1, \dots, a_n)$ . A solução consiste em uma permutação não decrescente  $A' = (a'_1, \dots, a'_n)$ . Apesar deste trabalho focar em números inteiros, a extensão para racionais, ponto flutuante e cadeias de caracteres tende a ser direta.

Todos os algoritmos de ordenação apresentam características que os fazem mais ou menos competitivos com relação a outros. Algumas destas características são o tipo de ordenação, estável ou não estável, utilização de espaço extra para a execução do algoritmo, tempo de ordenação. Alguns algoritmos podem ser mais rápidos que outros dependendo do tipo de entrada de dados. Por exemplo, a ordenação por seleção tende a ser vantajosa quando  $n$  é pequeno. A ordenação por inserção tende a ser rápida quando o vetor está parcialmente ordenado. A ordenação por contagem é vantajosa quando a diferença entre o maior e menor elemento é limitada.

Os algoritmos de ordenação mais conhecidos são baseados em comparações, como o *Merge Sort*, *Heap Sort*, *Insertion Sort* e *Quick Sort* e os baseados em contagem, como por exemplo *Counting Sort*, *Bucket Sort* e o *Radix*

*Sort.* Os algoritmos baseados em contagem necessitam de uma sequência de entrada com algumas restrições. Quando essas restrições são satisfeitas, esses algoritmos podem resolver o problema da ordenação em tempo linear.

Existe um limite de tempo inferior de  $\Omega(n \lg n)$  comparações para algoritmos de ordenação [8]. Este limite é baseado em uma árvore de decisão com  $n!$  folhas, cada uma representando uma permutação do vetor de entrada. Cada permutação é uma candidata a solução. Dado que uma comparação pode distinguir dois ramos de árvore, serão necessários, no mínimo,  $\lg(n!) = \Theta(n \lg n)$  comparações, no pior caso, para ordenar um vetor através de um algoritmo de ordenação baseado em comparações. Este limite inferior foi mal interpretado, gerando uma falsa crença em parte da comunidade de que ordenação é um problema  $\Omega(n \lg n)$ . Tal limite não se aplica, por exemplo, a algoritmos que usam outras operações além de comparações durante a ordenação. A ordenação por contagem é capaz de ordenar um vetor sem realizar nenhuma comparação entre elementos de  $S$ .

O algoritmo analisado neste trabalho é baseado em comparações e faz  $\Theta(n \lg n)$  comparações, porém são comparados  $(\lg n)^{1/5}$  números em  $O(1)$ . Ou seja, são efetuadas múltiplas operações em tempo constante. O parágrafo a seguir foi extraído de [5]:

“O caso de ordenar  $n$  inteiros de  $w$  bits no tempo  $o(n \lg n)$  foi considerado por muitos pesquisadores. Vários resultados positivos foram obtidos, cada um sob hipóteses um pouco diferentes a respeito do modelo de computação e das restrições impostas sobre o algoritmo. Todos os resultados pressupõem que a memória do computador está dividida em palavras endereçáveis de  $w$  bits. [6] introduziram a estrutura de dados de Árvore de Fusão e a empregaram para ordenar  $n$  inteiros no tempo  $O(n \lg n / \lg \lg n)$ . Esse limite foi aperfeiçoado mais tarde para o tempo  $O(n \sqrt{\lg n})$  por [3]. Esses algoritmos exigem o uso de multiplicação e de várias constantes pré-calculadas. [4] mostraram como ordenar  $n$  inteiros no tempo  $O(n \lg \lg n)$  sem usar multiplicação, mas seu método exige espaço de armazenamento que pode ser ilimitado em termos de  $n$ . Usando-se o hash multiplicativo, é possível reduzir o espaço de armazenamento necessário para  $O(n)$ , mas o limite  $O(n \lg \lg n)$  do pior caso sobre o tempo de execução se torna um limite de tempo esperado. Generalizando as árvores de pesquisa exponencial de [3], [10] forneceu um algoritmo de ordenação de tempo  $O(n(\lg \lg n)^2)$  que não usa multiplicação ou aleatoriedade, e que utiliza espaço linear. Combinando essas técnicas com algumas idéias novas, [7] melhorou o limite para ordenação até o tempo  $O(n \lg \lg n \lg \lg \lg n)$ . Embora esses algoritmos sejam inovações teóricas importantes, todos eles são bastante complicados e neste momento parece improvável que venham a competir na prática com algoritmos de ordenação existentes”.

Resultados: O algoritmo de ordenação  $O(n \lg n / \lg \lg n)$  analisado neste trabalho é conhecido na literatura. Nossa contribuição consiste em detalhar a estrutura de dados Árvore de Fusão e o algoritmo de ordenação  $O(n \lg n / \lg \lg n)$  proposto por [6].

## 1.1 Modelo Computacional

Considere um computador que trabalhe com palavras de  $w$  bits. Este computador é capaz realizar operações elementares como soma, subtração, multiplicação, divisão e resto de inteiros com  $w$  bits em tempo constante. Um computador de 64 bits tem capacidade de processar 64 bits em tempo constante, por exemplo.

O caso geral da ordenação trata de inteiros com precisão arbitrária. Para um inteiro com  $mw$  bits são necessários  $m$  acessos à memória antes de ler o número por completo. Este trabalho trata do caso restrito da ordenação onde os números são inteiros com  $w$  bits. Tais números estão no intervalo  $\{-2^{w-1}, \dots, 2^{w-1}\}$  armazenados como inteiros binários, com 1 bit para o sinal.

Este trabalho considerou um modelo computacional capaz de ler e escrever qualquer posição de memória em tempo constante, conhecido como memória RAM. O modelo de uma memória RAM é aceitável, apesar da coexistir com o modelo de memória de acesso sequencial. Neste último, é necessário movimentar a fita até a posição desejada antes da leitura, gastando tempo linear para ler um inteiro. O algoritmo *Merge Sort* é famoso por manter a complexidade  $O(n \lg n)$  mesmo num modelo de memória sequencial.

É razoável assumir que o computador é capaz de processar  $w = \log n$  bits em tempo constante. Se temos  $n$  inteiros de  $w$  bits na memória, o maior endereço de memória terá pelo menos  $\lg n$  bits. Assumir que é possível acessar qualquer posição de memória em tempo constante equivale ao computador processar endereços de  $\lg n$  bits em tempo constante. Observe que o número de bits do problema é  $nw \geq n \lg n$ . Isto significa que uma operação para cada bit é  $\Omega(n \lg n)$ .

Para melhor compreensão da ordenação, primeiro será mostrado como ordenar  $n$  números utilizando a estrutura de uma árvore B. A Árvore de Fusão, proposta por [6], é uma árvore B modificada.

## 2 Árvores B

Árvores B são árvores de pesquisa balanceadas com grau  $t$ , onde  $\frac{B}{2} \leq t \leq B$ , para  $B$  constante. Cada nó contém no mínimo  $\frac{B}{2}$  e no máximo  $B$  filhos, com exceção das folhas, que não contém nenhum filho e do nó raiz, que não apresenta restrição de número mínimo de filhos. Cada nó da árvore possui no mínimo  $\frac{B}{2} - 1$  chaves e no máximo  $B - 1$  chaves ordenadas, e todas as folhas se encontram na mesma altura. Veja que uma nó com grau 4 possui 3 chaves. A Figura 1 ilustra uma árvore B completa, onde todos os nós possuem  $B - 1$  chaves.

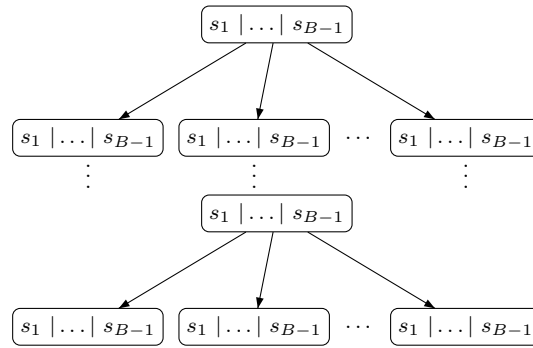


Figura 1: Estrutura de uma árvore B completa.

Além disso, a árvore B respeita a seguinte propriedade: cada nó não raiz possui  $t$  elementos ordenados  $S = (s_1, \dots, s_t)$ . Cada nó não folha e não raiz possui  $t + 1$  filhos  $(f_0, \dots, f_t)$  onde cada filho é uma árvore B. Os

elementos na árvore  $f_0$  são menores que  $s_1$ . Os elementos em  $f_i$  são maiores que  $s_i$  e menores que  $s_{i+1}$ . Os elementos em  $f_t$  são todos maiores que  $s_t$ . Veja Figura 2.

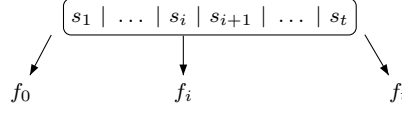


Figura 2: Estrutura de um nó de uma árvore B

Para encontrar uma chave  $k \notin S$  da árvore, é preciso determinar em qual filho do nó  $X$  continuar pesquisa. Se  $k < s_1$ , a procura continua no nó filho  $f_0$ . Se  $k > s_t$ , a procura continua no nó filho  $f_t$ . Se  $s_i < k < s_{i+1}$ , a procura continua no filho  $f_i$ , entre as chaves  $s_i$  e  $s_{i+1}$  de  $S$ .

Os tempos das operações de uma árvore B estão diretamente ligadas à sua altura. O seguinte lema baseia-se em [5].

**Lema 1** *Uma árvore B de grau  $B \geq 4$  e altura  $h$  respeita a seguinte relação:*

$$h = O(\log_B n)$$

Para buscar uma chave  $k$  em uma árvore B, deve-se, no pior caso, procurar sequencialmente  $B - 1$  chaves em cada nó e repetir o processo em cada nível. Ou seja, o tempo de pesquisa de uma árvore B é de  $O(B \log_B n)$ . Supondo que  $B$  é constante, a pesquisa se dá em  $\Theta(\lg n)$ .

Para se inserir uma chave  $k$  na árvore B, primeiro deve pesquisar o nó em que a nova chave será inserida. Se houver espaço no nó, gasta-se mais  $O(B)$  para acomodar a chave no nó. Este é o custo para inserir um elemento em uma posição central de um vetor de tamanho  $B$ . Se o nó que receberá a chave estiver lotado, ele deve ser dividido ao meio. Veja Figura 3. Essa separação acontece da seguinte maneira: a chave mediana  $s_m$  do nó em questão vai para o nó pai. São criados dois nós, um com as chaves  $s_i < s_m$  e outro com as chaves  $s_i > s_m$ . Tais nós passam a ser filhos à esquerda e à direita de  $s_m$ , cada um com exatamente  $\frac{B}{2} - 1$  chaves.

O custo de dividir um nó ao meio é  $O(B)$  através de operações elementares em vetores. Pode-se perceber que o nó pai pode ser um nó completo também. Neste caso, o processo de separação se propaga para cima da árvore, possivelmente até a raiz.

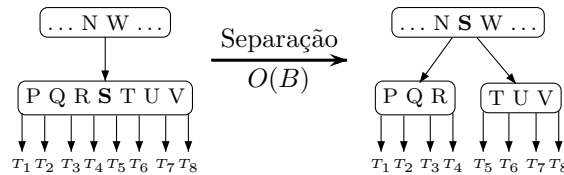


Figura 3: Inserção de uma chave em um nó completo da árvore B. Fonte: [5].

Se esse processo se propagar até a raiz da árvore a separação será realizada em tempo  $O(Bh) = O(B \cdot \log_B n)$ . Esta complexidade de pior caso, entretanto, pode melhorar através de uma análise amortizada. Cada separação

custa  $O(B)$  e cria um novo nó. Como, ao final da inserção de  $n$  elementos teremos no máximo  $1 + \frac{n-1}{B/2-1}$  nós, o custo de todas as separações será  $O(B(1 + \frac{n-1}{B/2-1})) = O(n)$ . Ou seja, após a inserção de  $n$  elementos, será gasto  $O(n)$  para realizar todas as separações.

Removendo-se o custo da separação, cada inserção levará  $O(B \log_B n + B)$  que é o custo para se encontrar o nó mais o custo para se inserir uma chave em um nó.

Para ordenar uma sequência de  $n$  números inteiros utilizando uma árvore B, é necessário inserir todos os elementos em uma árvore inicialmente vazia. Ao final, tem-se um árvore B com os  $n$  elementos da sequência inicial. Para se obter a sequência ordenada, é realizado um percurso em ordem na árvore, como demonstrado na Figura 4. As setas contínuas representam para onde caminhar na árvore, e as setas pontilhadas representam a leitura da chave na sequência. Cada seta possui um número, que indica a sequência dos elementos visitados no percurso em ordem.

O tempo para ordenarmos  $n$  números será então a soma dos tempos de inserção das  $n$  chaves, ou seja,  $O(n \cdot B \log_B n + nB)$ . Se B for constante, a complexidade final será  $O(n \lg n)$ .

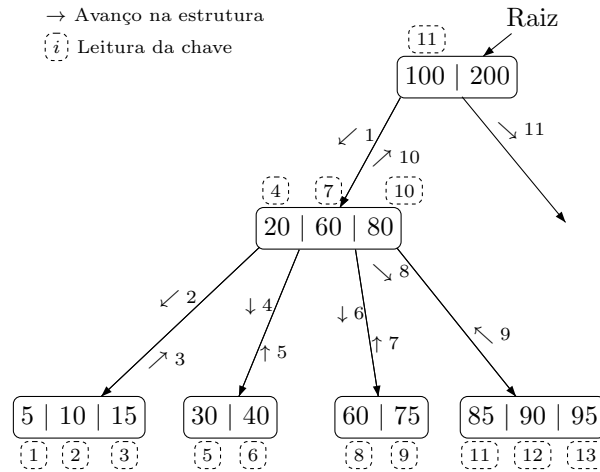


Figura 4: Visitação em ordem de uma árvore B

### 3 Árvores de Fusão

Nesta seção será descrita a estrutura Árvore de Fusão proposta por [6]. Uma Árvore de Fusão é semelhante a uma árvore B. Seja  $t$  o grau em um nó de uma árvore B. Considerando-se um nó não raiz e não folha, o valor de  $t$  está entre  $\frac{B}{2}$  e B. O mesmo vale para a estrutura de dados Árvore de Fusão. Entretanto, o valor de B na Árvore de Fusão não é constante mas sim uma função de  $n$ . Mais precisamente  $B = (\lg n)^{\frac{1}{5}}$ . Outra diferença entre árvore B e Árvore de Fusão é que na primeira gasta-se B para fazer a busca de uma chave  $k$  dentro de um nó, enquanto numa Árvore de Fusão, a busca por uma chave é feita em  $O(1)$ .

Considere o problema do sucessor/predecessor de uma chave  $x$  em um conjunto  $S$ . Este problema consiste em encontrar o número imediatamente inferior/superior a  $x$  em  $S$ . Árvores de Fusão são estruturas de dados semelhantes a árvores B, mas que conseguem resolver o problema predecessor/sucessor em tempo  $O(1)$  dentro de

um nó. Dada uma chave de pesquisa  $x$ , a Árvore de Fusão consegue achar o ramo filho relativo a  $x$  em tempo constante, apesar de possuir um número de filhos crescente em relação a  $n$ .

A notação a seguir é necessária para o estudo de Árvore de Fusão [6]:

**Definição 1**  $\text{rank}(x)$ : Dado um conjunto  $S$  de números inteiros e um número  $x$ , denota-se  $\text{rank}(x)$  o valor  $|\{t \mid t \in S, t \leq x\}|$ . Em outras palavras,  $\text{rank}(x)$  representa a quantidade de números em  $S$  menores ou iguais a  $x$ .

Veja que o problema de ordenar  $n$  números pode ser resumido a encontrar  $\text{rank}(x)$ , pois essa função fornece a posição exata de  $x$  no vetor ordenado. Além disso,  $\text{rank}(x)$  indica o nó filho para continuar a busca de um elemento  $x$  em um nó de uma árvore B. Como a Árvore de Fusão baseia-se na estrutura de dados *trie* do trabalho [2], tal estrutura será descrita a seguir.

### 3.1 Estrutura de Dados *trie* de [2]

Seja uma *trie* uma árvore Binária construída da seguinte forma. Dado um número binário  $x$  com  $w$  bits, cada bit de  $x$  é um nó na *trie*. Se o bit mais significativo de  $x$  for zero,  $x$  é um filho à esquerda da raiz. Se for 1,  $x$  é um filho à direita da raiz. Esta propriedade vale para os filhos, a partir do próximo bit.

Dado um inteiro qualquer, seja  $b_i$  seu  $i$ -ésimo bit menos significativo. Assim  $b_0$  é o bit menos significativo,  $b_1$  o segundo bit menos significativo e assim por diante. Considere dois números binários  $s_1 = 11101001$  e  $s_2 = 11111001$ . A Figura 5 apresenta uma *trie* contendo  $s_1$  e  $s_2$ . Note que as folhas da *trie* estão sempre ordenadas. Suponha também que o  $\text{rank}(x)$  está calculado para todos os elementos da *trie*.

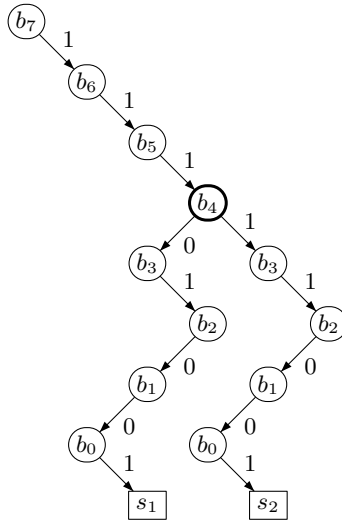


Figura 5: Estrutura *trie* para  $s_1$  e  $s_2$ . Basta analisar o  $b_4$  para ordená-los

Considere a seguinte definição:

**Definição 2**  $\Delta(s_1, s_2)$ : Dados dois inteiros  $s_1$  e  $s_2$ , seja  $\Delta(s_1, s_2)$  o bit de interesse entre  $s_1$  e  $s_2$ , ou seja, o bit mais significativo que  $s_1$  diverge de  $s_2$ .

Para comparar números binários, não é necessário comparar todos os bits, e sim os **bits de interesse**, que são os bits mais significativos que diferenciam um conjunto de números. Por exemplo, para compararmos os números binários  $s_1 = 11101001$  e  $s_2 = 11111001$ , basta comparar o bit mais significativo que diverge em  $s_1$  e  $s_2$ . Nesse exemplo, o bit que se deve comparar é o  $b_4$ , que em  $s_1$  é 0 e em  $s_2$  vale 1. Sendo assim,  $\Delta(s_1, s_2) = b_4$ . A partir disso, se sabe que o número  $s_2$  é maior que  $s_1$  sem analisar o restante dos bits. Veja Figura 6.

Seja  $\oplus$  um xor bit a bit entre duas palavras. Dados dois inteiros  $s_1$  e  $s_2$ ,  $\Delta(s_1, s_2)$  pode ser obtido pela seguinte fórmula:

$$\Delta(s_1, s_2) = \lfloor \lg(s_1 \oplus s_2) \rfloor.$$

Considere uma sequência de inteiros  $S = (s_1, \dots, s_n)$  e uma estrutura de dados *trie*. Após inserir todos os elementos de  $S$  na *trie*, será feita uma compressão onde serão guardados apenas os bits de interesse. Esta nova árvore será chamada de *trie* condensada

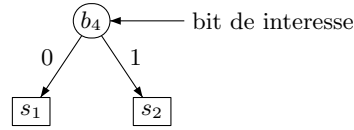


Figura 6: Estrutura *trie* condensada para comparar  $s_1$  e  $s_2$  com somente o bit de interesse

A *trie* condensada guarda cada elemento de  $S$  em uma folha. Já os nós internos da *trie* condensada guardam o bit de interesse para comparação entre seus dois filhos.

**Lema 2** Dada uma *trie* condensada contendo  $S = (s_1, \dots, s_t)$ , o número de bits de interesse será menor igual a  $t - 1$ .

O Lema 2 decorre do fato de cada bit de interesse criar uma ramificação na *trie*. O número de ramificações será exatamente  $t - 1$ . Eventualmente, duas ramificações distintas podem ocorrer no mesmo bit de interesse.

Para buscar  $x$  em uma *trie* condensada, são comparados os bits de  $x$  com os bits da *trie* da raiz até a folha. Em cada nó, caso o bit de  $x$  seja 0, a busca continua no filho à esquerda. Caso o bit seja 1, a busca continua no filho à direita. O respectivo bit considerado em cada nível está armazenado no nó da *trie* condensada. A Figura 7 ilustra uma pesquisa com o elemento  $x$  em uma *trie* condensada contendo os elementos  $a, b, c$  e  $d$ . Seja  $\text{BuscaTrie}(x)$  o resultado desta busca. Na Figura 7,  $\text{BuscaTrie}(x) = c$

Ao realizar esse procedimento, chega-se a uma folha, que é um elemento  $s$  de  $S$ . Isso mostra que  $x$  e  $s$  têm os mesmos bits nas posições percorridas no caminho da *trie*.

No exemplo da Figura 7 nota-se que o primeiro bit a divergir entre  $x$  e  $c$  é o  $b_2$ , ou seja  $\Delta(x, c) = b_2$ . A Figura 8 mostra como a *trie* fica após a inserção do elemento  $x$  na estrutura.

### 3.1.1 Calculando o $\text{rank}(x)$

Suponha um conjunto  $S = (s_1, \dots, s_t)$  inseridos numa *trie* condensada. Esta seção mostrará como calcular  $\text{rank}(x)$  para uma dada chave  $x$ . Primeiro, calcula-se  $s' = \text{BuscaTrie}(x)$ . O elemento  $s'$  possui valores igual a  $x$  nos bits de

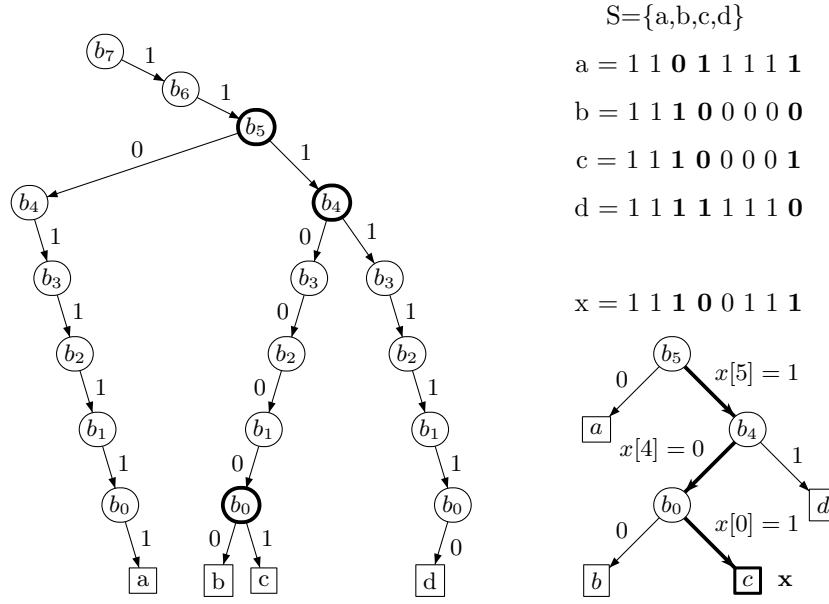


Figura 7: Pesquisa de uma chave  $x$  na estrutura de dados *trie* condensada.

interesse. Se  $s'$  for igual a  $x$  nos demais bits,  $\text{rank}(x) = \text{rank}(s') + 1$  e o problema se encerra. Caso contrário, será necessária uma segunda busca. Será calculado o bit de interesse  $b' = \Delta(x, s')$ .

**Lema 3** *O bit de interesse  $b' = \Delta(x, s')$  é o novo bit de interesse da trie condensada com  $S \cup \{x\}$ .*

Considere dois casos. No primeiro o bit  $b'$  de  $x$  vale 1, ou seja  $x[b'] = 1$  enquanto no segundo caso  $x[b'] = 0$ .

**Lema 4** *Os bits mais significativos de  $x$  e  $s'$  são idênticos. O primeiro bit a divergir é  $b'$ . Considere a ramificação entre  $x$  e  $s'$  na trie contendo  $S \cup \{x\}$ . Se  $x[b'] = 1$ , o predecessor de  $x$  é o maior elemento no ramo  $b' = 0$ . Se  $x[b'] = 0$ , o sucessor de  $x$  é o menor elemento no ramo  $b' = 1$ .*

**Caso a** ( $x[b'] = 1$ ) No exemplo da Figura 9, verifica-se que o predecessor de  $x$  é o maior elemento da sub-árvore em destaque.

Para encontrar o  $\text{rank}(x)$ , é necessário realizar uma segunda busca. Do bit mais significativo até o bit de interesse  $b'$ , percorre-se a *trie* condensada usando os bits de  $x$ . A partir do bit de interesse, é necessário encontrar o maior elemento da subárvore, ou seja, é necessário descer na árvore sempre para a direita, em direção ao maior elemento.

Para se obter este comportamento, será criada uma segunda chave de busca,  $x'$  da seguinte maneira:

$$\begin{array}{r}
 x = x_{w-1}x_{w-2} \dots x_2x_1x_0 \\
 \text{OR} \quad \quad \quad 1\ 1\ 1\ 1 \\
 \hline
 x' = x_{w-1}x_{w-2} \dots 1\ 1\ 1\ 1
 \end{array}$$

O número de 1's no final de  $x'$  é  $b'$ . Ao trocar um bit de  $x$  por 1 a partir do bit de interesse, cria-se uma chave de busca que irá encontrar o predecessor de  $x$ . Seja  $s'' = \text{BuscaTrie}(x')$ . Então  $\text{rank}(x) = \text{rank}(s'') + 1$ . Veja um exemplo na Figura 10. Observe que a máscara é computável em  $O(1)$  a partir de  $2^{b'+1} - 1$ .



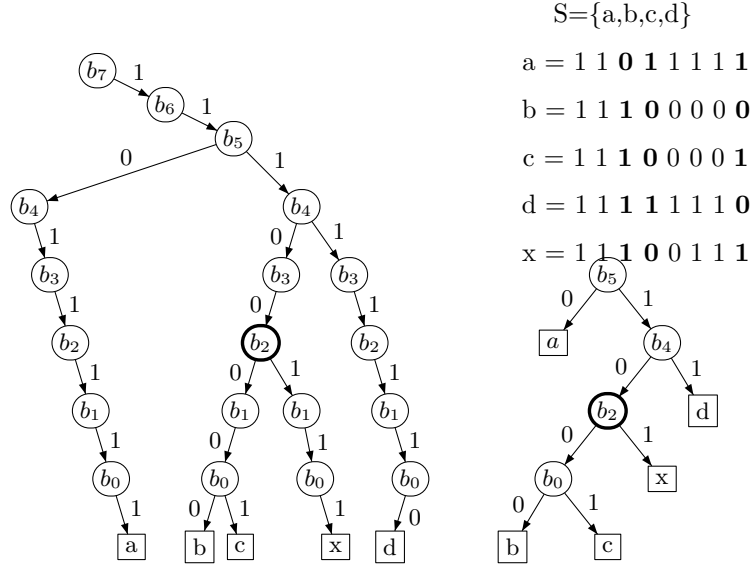


Figura 8: *trie* e *trie* condensada após inserção do elemento  $x$  [2].

**Caso b** ( $x[b'] = 0$ ) No exemplo da Figura 11, verifica-se que o sucessor de  $x_3$  é o menor elemento da sub-árvore em destaque. Para encontrar o  $\text{rank}(x_3)$ , é necessário realizar uma segunda busca. Do bit mais significativo até o bit de interesse  $b'$ , percorre-se a *trie* condensada usando os bits de  $x_3$ . A partir do bit de interesse, continua-se descendo na árvore sempre para a esquerda, em direção ao menor elemento.

Para obter este comportamento, será criada uma segunda chave de busca,  $x'$  da seguinte maneira:

$$\begin{array}{r}
 x = x_{w-1}x_{w-2} \dots x_2x_1x_0 \\
 \text{AND} \quad 1 \ 1 \ 1 \ 1 \dots 0 \ 0 \ 0 \ 0 \\
 \hline
 x' = x_{w-1}x_{w-2} \dots 0 \ 0 \ 0 \ 0
 \end{array}$$

O número de 0's no final de  $x'$  é  $b'$ . Ao trocar um bit de  $x$  por 0 a partir do bit de interesse, cria-se uma chave de busca que irá encontrar o predecessor de  $x$ . Seja  $s'' = \text{BuscaTrie}(x')$ . Então  $\text{rank}(x) = \text{rank}(s'')$ . Veja um exemplo na Figura 12.

### 3.2 Característica da Árvore de Fusão

Basicamente, uma Árvore de Fusão é uma árvore B com grau  $B = (\lg n)^{\frac{1}{5}}$ , ou seja, o grau é crescente com relação à quantidade de números de itens a serem ordenados, como exemplificado na Figura 13.

Como a altura de uma árvore B é proporcional a  $\log_B n$  e na árvore de fusão  $B = (\lg n)^{\frac{1}{5}}$  então a altura  $h$  é O de:

$$\log_B n = \frac{\lg n}{\lg B} = \frac{\lg n}{\lg (\lg n)^{1/5}} = \frac{\lg n}{\lg^{\frac{1}{5}} \lg n} = O\left(\frac{\lg n}{\lg \lg n}\right)$$

O tempo de busca dentro de um nó de uma árvore B é  $O(B)$ , pois é feita uma busca sequencial para encontrar o filho. Isso em cada nível da árvore, resultando em  $O(B \log_B n)$ . Numa Árvore de Fusão, o nó filho é encontrado em  $O(1)$ . Assim, a busca termina em  $O(\log_B n)$ . Como  $B = (\lg n)^{\frac{1}{5}}$ , a busca ocorre em

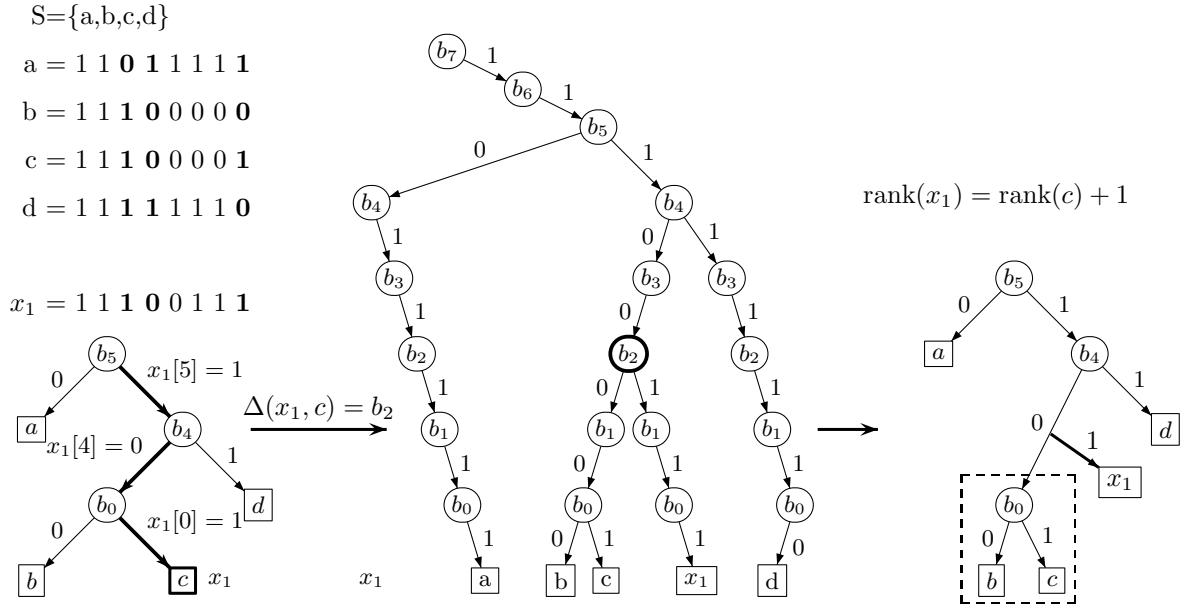


Figura 9: Obtenção do  $\text{rank}(x_1)$

$$\log_B n = O\left(\frac{\lg n}{\lg \lg n}\right)$$

Como visto anteriormente, para ordenar números binários, não é necessário comparar todos os bits, mas somente os bits de interesse. Esses bits de interesse são chamados de sketch:

**Definição 3**  $\text{sketch}(s)$ : A *sketch* de uma palavra  $s$  consiste em descartar todos os seus bits, exceto os bits de interesse. A ordem das palavras são preservadas, ou seja,  $s_i < s_j$  se e somente se  $\text{sketch}(s_i) < \text{sketch}(s_j)$ .

Por exemplo, na Figura 7, os *sketches* dos elementos  $a$ ,  $b$ ,  $c$  e  $d$  serão respectivamente 011, 100, 101 e 110. E pode-se perceber que a ordem entre os *sketches* não muda em relação aos números originais.

A ideia central da Árvore de Fusão está em como ela armazena as chaves em cada nó. Cada um de seus nós possui  $t \leq B - 1 = O(w^{\frac{1}{B}})$  chaves. Conforme o Lema 2, em um conjunto de  $B - 1$  chaves, a *trie* terá, no máximo,  $B - 2$  bits de interesse. Um nó possui no máximo  $B - 1$  *sketches*, cada *sketch* possui no máximo  $B - 2$  bits de interesse. Então, a soma dos bits dos *sketches* será:

$$(B - 1) \cdot (B - 2) \leq w^{\frac{1}{B}} \cdot w^{\frac{1}{B}} = O(w^{\frac{2}{B}}) = o(w) = o(\lg n).$$

Concluimos, então, que a soma dos bits dos *sketches* das chaves em um nó cabem em apenas uma palavra da memória. Cada nó da árvore conterá uma palavra que armazenará todos os *sketches* das chaves e mais alguns bits, conforme a definição a seguir:

**Definição 4** *Nó Sketch*: *Nó Sketch* é um nó que contém todos os *sketches* das chaves  $(s_1, \dots, s_t)$ . Estes *sketches* podem ser armazenados em uma única palavra, sendo que cada *sketch* é precedido de um bit separador com valor 1.

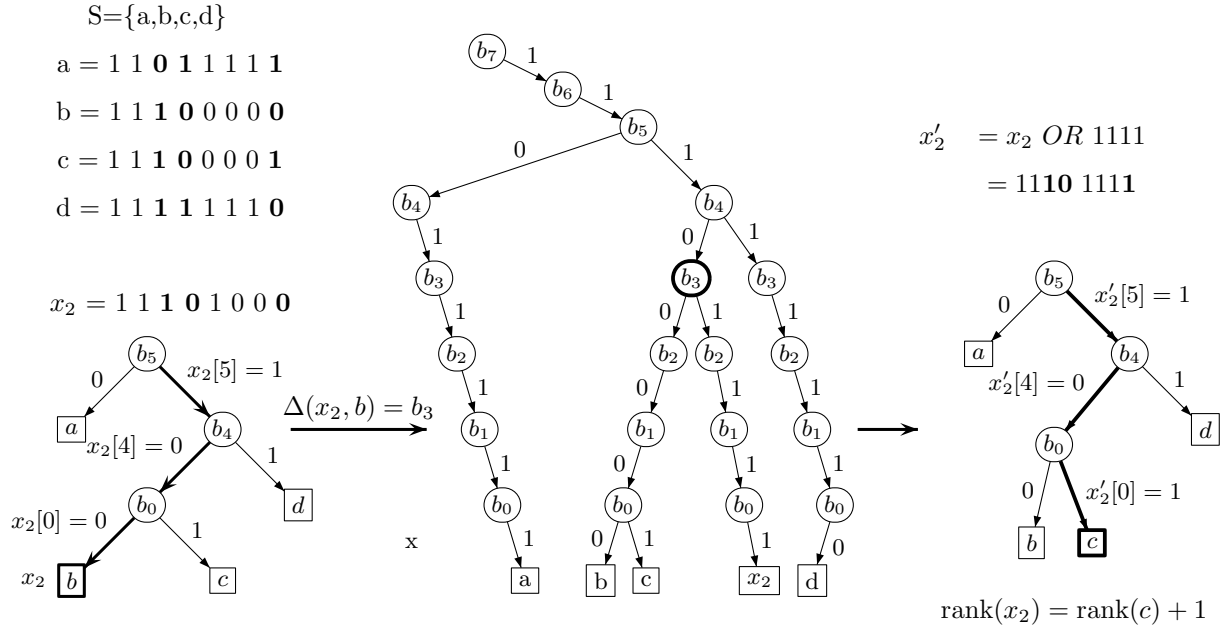


Figura 10: Obtenção do  $\text{rank}(x_2)$  após a segunda busca na *trie* condensada

O Nó Sketch será a concatenação dos sketches das chaves:

$$w_{\text{node}} = \text{lsketch}(s_1)\text{lsketch}(s_2)\dots\text{lsketch}(s_t).$$

Além disso, sketches são concatenados em ordem crescente.

Na próxima seção será mostrado a comparação de um dado número  $x$  com todas as chaves de um nó em tempo constante, baseado em [9].

### 3.3 Múltiplas Comparações em tempo constante

Considere um nó da árvore de fusão com elementos  $S = (s_1, \dots, s_t)$ . Suponha que os bits de interesse deste conjunto sejam  $(i_1, \dots, i_{t'})$  com  $t' < t$ . Para comparar uma chave de pesquisa  $x$  com todas as chaves de um nó, primeiramente computa-se  $\text{sketch}(x)$ .

Para extrair o bit de interesse  $i_1$  e colocá-lo na posição 0 de um  $\text{sketch}(x)$ , primeiro é feito um *AND* bit a bit de  $x$  e uma máscara contendo zeros e um único 1 na posição  $i_1$ . Uma vez aplicada esta máscara, é necessário mover o bit para a posição 0. Isso pode ser feito por um deslocamento *delta* para a esquerda, obtida por uma multiplicação por  $2^{\text{delta}}$ .

Para se obter todos os bits do  $\text{sketch}(x)$  é necessário fazer o *AND* bit a bit de  $x$  com uma máscara contendo valores 1 apenas nos bits de interesse  $(i_1, \dots, i_{t'})$ . Esta máscara será construída junto com a *trie* condensada e estará disponível no momento da busca de  $x$ .

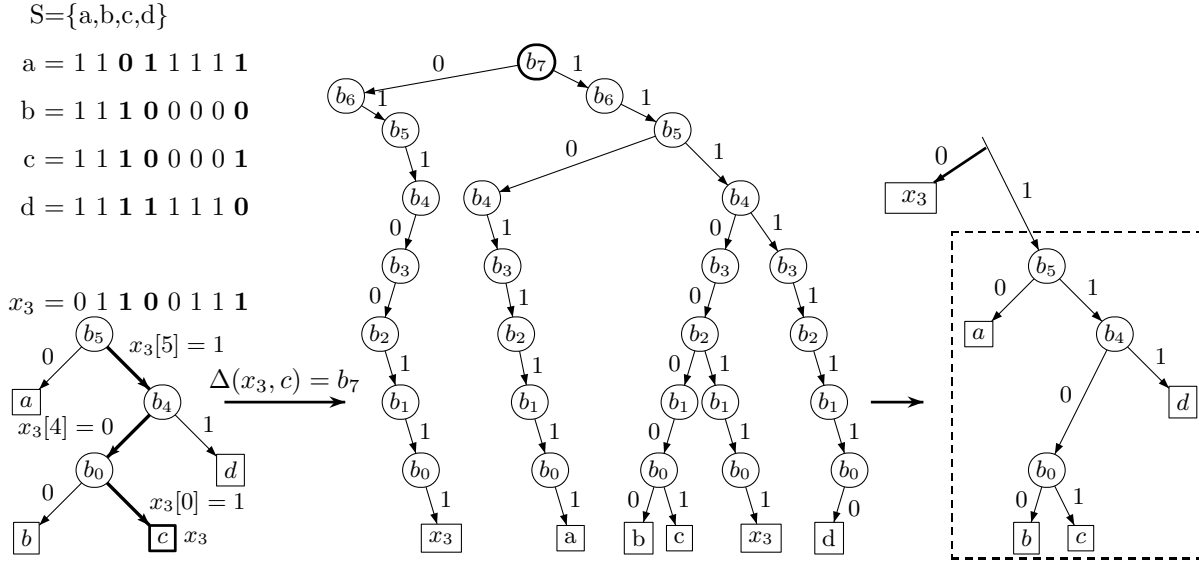
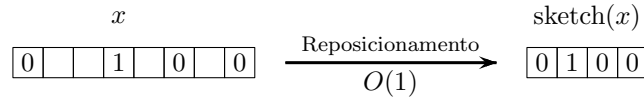


Figura 11: Obtenção do  $\text{rank}(x_3)$ . O menor elemento da sub-árvore em destaque é o sucessor de  $x_3$ .

Após aplicar a máscara, deve-se mover os bits para as posições iniciais criando o  $\text{sketch}(x)$  em  $O(1)$  conforme a figura abaixo:



Ao multiplicar um inteiro  $x$  por uma constante pré-definida, é possível alterar a posição dos bits de  $x$ . Obter este reposicionamento de bits com uma única multiplicação não é tarefa trivial. O trabalho [1] mostra a existência de constantes pré-definidas que obtêm um reposicionamento dos bits. O resultado não é perfeito, pois as constantes geram alguns zeros adicionais no sketch. Tais zeros são criados de forma a não alterar o funcionamento do algoritmo.

Uma vez obtido o  $\text{sketch}(x)$ , seu valor será replicado dentro de uma palavra acrescido do bit separador 0 da seguinte maneira:

$$w_x = 0 \text{ sketch}(x) \ 0 \text{ sketch}(x) \ \dots \ 0 \text{ sketch}(x)$$

Suponha que  $\text{sketch}(x)$  possui 6 bits, assim tem-se que

$$\begin{aligned} w_x &= \text{sketch}(x) + \text{sketch}(x) \cdot 2^7 + \text{sketch}(x) \cdot 2^{14} + \dots \\ &= \text{sketch}(x) \cdot (\dots 10000010000001). \end{aligned}$$

Ou seja, é possível calcular  $w_x$  a partir de  $\text{sketch}(x)$  com uma única multiplicação.

**Fato 1** Ao subtrair  $1\text{sketch}(s_i) - 0\text{sketch}(x)$ , o resultado começará com 1 se e somente se  $\text{sketch}(x) \leq \text{sketch}(s_i)$ .

Seja  $\text{sketch}(x) = 1111$  e  $\text{sketch}(s_i) = 0000$ , assim,  $1\text{sketch}(s_i) - 0\text{sketch}(x) = 10000 - 01111 = 00001$ . Como a subtração começa com zero, então  $\text{sketch}(x) > \text{sketch}(s_i)$ .

Agora, se  $\text{sketch}(x) = 0000$  e  $\text{sketch}(s_i) = 00001$ , tem-se  $1\text{sketch}(s_i) - 0\text{sketch}(x) = 10001 - 00000 = 10001$ . Como o primeiro bit é 1,  $\text{sketch}(x) \leq \text{sketch}(s_i)$ .

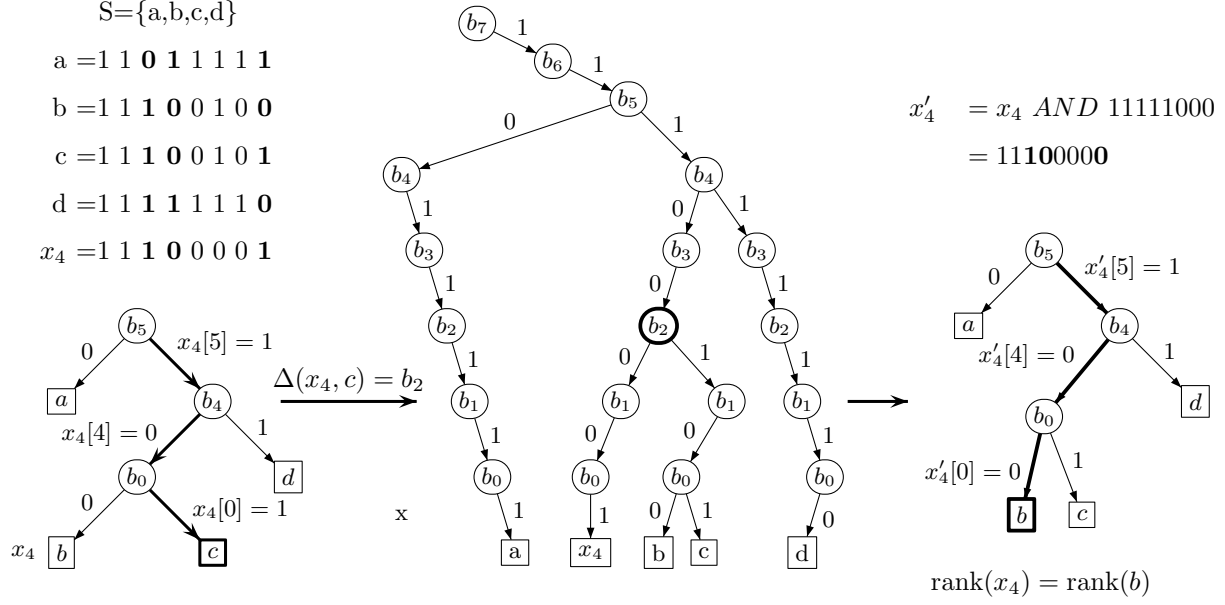


Figura 12: Obtenção do  $\text{rank}(x_4)$  após a segunda busca na *trie* condensada

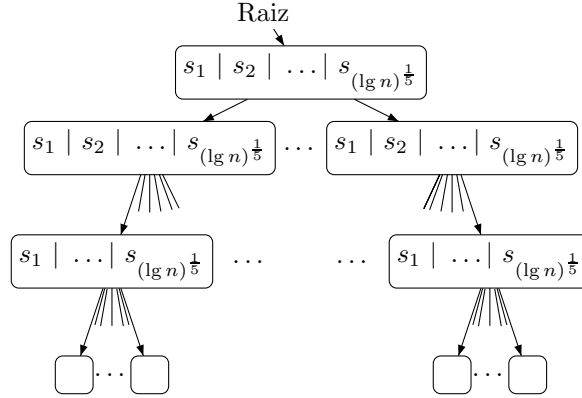


Figura 13: Estrutura de uma Árvore de Fusão Completa.

Para comparar  $x$  com todas as chaves de  $S$  em  $O(1)$ , é necessário calcular a subtração entre  $1\text{sketch}(s) - 0\text{sketch}(x)$  para todos  $s \in S$  em uma única operação. Ou seja, múltiplas comparações em uma única operação. Calcula-se

$$w_{res} = w_{node} - w_x.$$

O primeiro bit de cada bloco indicará se  $\text{sketch}(x)$  é menor igual ou maior que  $\text{sketch}(s_i)$ . Vale a pena lembrar que os *sketches* estão ordenados dentro do *nó sketch*  $w_{node}$ . É necessário encontrar o primeiro bit de cada bloco que valha 1. Suponha que o tamanho do bloco  $0\text{sketch}(x)$  é  $r$ . Para remover todos os bits, exceto os primeiros bits de cada bloco, será feita uma operação *AND bit-a-bit* entre  $w_{res}$  e uma máscara contendo 1 somente nas posições de interesse,  $(r, 2r, \dots)$ . Seja  $w'_{res}$  o resultado deste AND bit a bit.

O próximo passo é encontrar o bit mais significativo igual a 1. Esta operação equivale ao cálculo de  $\lfloor \lg(w'_{res}) \rfloor$  e precisa ser realizada em  $O(1)$ . A solução deste problema é conhecida na literatura [11].

O elemento  $s = \text{BuscaTrie}(x)$  pode ser computado diretamente a partir da posição do primeiro 1 no início de um bloco em  $w_{res}$ . Seguindo os passos da sessão anterior tem-se o  $\text{rank}(x)$  em  $O(1)$ . Usando esta operação, descobre-se qual filho seguir na busca dentro da Árvore de Fusão em  $O(1)$ .

Como exemplo, será calculado  $\text{BuscaTrie}(x)$ , com os mesmos valores da Figura 7.

$$S = (a, b, c, d) = (11011111, 11100000, 11100001, 11111110) = (223, 224, 225, 245)$$

Considere que  $x = 11100111 = 231$  para esse caso. Os *sketches* dos elementos serão

$$\text{sketch}(a) = 011; \quad \text{sketch}(b) = 100;$$

$$\text{sketch}(c) = 101; \quad \text{sketch}(d) = 110$$

o nó *sketch* será

$$w_{node} = 1\ 011\ 1\ 100\ 1\ 101\ 1\ 110 = 48350$$

e o  $\text{sketch}(x)$  será 101. A palavra para subtrair de  $w_{node}$  será:

$$w_q = 0\ 101\ 0\ 101\ 0\ 101\ 0\ 101 = 21845$$

Subtraindo os valores e aplicando a função *AND* explicada acima, obtém-se

$$\begin{aligned} w_{res} &= (w_{node} - w_q) \text{ AND } 1\ 000\ 1\ 000\ 1\ 000\ 1\ 000 \\ &= (48350 - 21845) \text{ AND } 1\ 000\ 1\ 000\ 1\ 000\ 1\ 000 \\ &= 26505 \text{ AND } 1\ 000\ 1\ 000\ 1\ 000\ 1\ 000 \\ &= 0\ 110\ 0\ 111\ 1\ 000\ 1\ 001 \text{ AND } 1\ 000\ 1\ 000\ 1\ 000\ 1\ 000 \\ &= 0\_0\_1\_1\_1\_ = 136 \end{aligned} \tag{1}$$

O primeiro bit 1 é  $\lfloor \lg(136) \rfloor = b_7$ . Lembrando que os bits são zero indexados, tem-se 8 bits até o primeiro 1. Dividindo esse valor por 4, que é o tamanho do bloco, obtem-se  $= 2$ , que é o penúltimo elemento, ou seja o elemento  $\text{BuscaTrie}(x) = c$  pois  $S = (a, b, c, d)$

### 3.4 Ordenando em tempo $o(n \lg n)$

Já foi visto como realizar uma busca de uma chave com  $w$  bits em tempo  $O(\frac{\lg n}{\lg \lg n})$  utilizando uma Árvore de Fusão, e como ordenar  $n$  elementos utilizando uma árvore B comum. Então, para ordenar  $n$  números, basta se inserir elemento por elemento na árvore. [12] mostra como transformar uma Árvore de Fusão estática, analisada neste artigo, em dinâmica, ou seja, que aceite atualizações nas chaves, fazendo isso em tempo  $O(\frac{\lg n}{\lg \lg n} + \lg(\lg(n)))$  por atualização, obtendo assim uma ordenação

$$\Theta(n \cdot O(\log_w(n) + \lg(\lg(n))) \frac{\lg n}{\lg \lg n}) =$$

$$\Theta\left(n \cdot \frac{\lg n}{\lg \lg n}\right)$$

## 4 Conclusão

O objetivo deste trabalho foi explicar de maneira detalhada um algoritmo de ordenação que ordena  $n$  números em tempo  $O(\frac{n \lg n}{\lg \lg n})$ . Para alcançar tal objetivo, verificou-se pouco material disponível sobre o assunto, o que dificultou o desenvolvimento do trabalho, pois o algoritmo emprega várias definições, teoremas e ideias que não são triviais.

Este trabalho deixa algumas questões complexas em aberto: descobrir o primeiro  $b_1$  em uma palavra, calcular o nó *sketch* em tempo  $O(1)$ , utilizar Árvores de Fusão dinâmica para inserir os nós na árvore de interesse.

De qualquer forma, este trabalho conseguiu completar com êxito a análise detalhada da estrutura de dados da Árvore de Fusão, estrutura fundamental do primeiro algoritmo de ordenação  $o(n \lg n)$  e base para muitos algoritmos subsequentes.

Este trabalho também revela que os limites inferiores de tempo precisam ser vistos com muito cuidado. Se um problema qualquer precisa de, no mínimo,  $f(n)$  operações, o verdadeiro limite inferior é  $\Omega(f(n)/\lg n)$  pois os modelos computacionais aceitos tem a capacidade de lidar com  $\lg n$  de bits em tempo constante.

Um trabalho futuro interessante seria implementar o algoritmo de ordenação baseado em árvores de fusão e comparar seu desempenho com algoritmos tradicionais. Outro ponto que merece investigação é a possibilidade de se fazer múltiplas operações em  $O(1)$ . Dentro da teoria, poderia-se investigar quais outros problemas poderiam ter sua complexidade baixada por meio desta estratégia. Já em computação aplicada, o uso de múltiplas operações dentro de uma palavra pode acelerar algoritmos tradicionais.

## Referências

- [1] Lecture 12, 2012.
- [2] Miklós Ajtai, Michael L. Fredman, and János Komlós. Hash functions for priority queues. *Information and Control*, 63(3):217–225, December 1984.
- [3] Arne Anderson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37<sup>th</sup> Annual Symposium on Foundations of Computer Science*, pages 135–141, 1996.
- [4] Arne Anderson, Torben Hagerup, Stefan Nilsson, and Rajeev Ramam. Sorting in linear time? *Journal of Computer and System Sciences*, pages 57:74–93, 1998.
- [5] Thomas H. Cormen. *Algoritmos - Teoria e Prática*. Elsevier, 2001.
- [6] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, pages 47:424–436, 1993.
- [7] Yijie Han. Improved fast integer sorting in linear space. In *Proceedings of the 12<sup>th</sup> ACM-SIAM Symposium of Discrete Algorithms*, pages 793–796, 2001.

- [8] D. E. Knuth. *The Art of Computer Programming*, volume 3. Reading, 2 edition, 1998.
- [9] scribe: Nicholas Zehender Prof. Erik Demaine. Lecture 10. MIT - Massachusetts Institute of Technology, 3 2010. Advanced Data Structures.
- [10] Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9<sup>th</sup> ACM-SIAM Symposium of Discrete Algorithms*, pages 550–555, 1998.
- [11] Henry S. Warren. *Hacker's delight*. Pearson Education, 2003.
- [12] Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29:1030–1049, December 1999.